The quarterly newsletter for JSBSim, an open source flight dynamics model in C++

Volume 1, Issue 4

15 January 2005

Back of the Envelope

JSBSim Configuration File Format v2.0 Nears Release

In the past few months a major modification task has been ongoing around the JSBSim configuration file format. This was done in order to:

- Add new capabilities
- Allow specification of units
- Make the JSBSim "language" valid XML
- Take advantage of what XML could offer us
- Be more compatible with the emerging DAVE-ML standard
- Use an open source XML parser to offload parsing responsibilities from JSBSim

This effort has been mostly completed at this time. The Cessna 172x model has been migrated to the new format and tested. Data comparisons between old and new simulation runs (regression testing) show that – at least in this test case – there have been no unintended side effects of the changeover.

This change does not simply involve the JSBSim aircraft configuration file format. The engine, thruster, script, and reset files have also been modified. The automatic configuration file generating web application, "Aeromatic", has also been modified to produce config files in the new format, as has Bill Galbraith's modification to the DATCOM application, called DATCOM+.

Most importantly, to allow existing users to use their aircraft models using the new code, the older version of JSBSim has been modified to allow the standalone version of JSBSim (the application compiled with the JSBSim.cpp file) to now take another argument on the command line. For instance,

```
./jsbsim --aircraft=747 --initfile=reset00 --convert
```

The "convert" option directs JSBSim to output (to the standard output) a configuration file in the new v2.0 JSBSim config file format. This conversion capability has been completed and early testing shows that it works as expected. At this time the conversion program only converts the aircraft file itself, not any external files that might be referenced or used.

New Capability: Equation Handling

One of the most exciting new capabilities offered in the v2.0 config file format is the ability to specify equations. Now, instead of specifying a lookup table for propeller thrust, or lift coefficient one can specify an equation – or, rather, a function.

Here is how the lift due to alpha is now specified in v2.0 (see Fig. 1, next page). You will notice that the COEFFICIENT element is gone. Coefficients are still there, but what is actually being specified by the function definition in the example above is a force contribution. There will normally be many contributing elements in each axis that will comprise the total force (or moment) in an axis.

Looking closer at the example, one can see that there is a "product" element that contains property and table elements. The table element describes the actual lift coefficient due to alpha, with lookup indices into the table of alpha and a stall hysteresis parameter (which we don't have to know much about for this discussion). In order to turn the coefficient into an actual force, we have to multiply the coefficient by the non-dimensionalizing quantities qbar and reference area. With the new version of JSBSim, we have created a new prop-

(Continued on page 2)

One of the most exciting new capabilities offered in the v2.0 config file format is the ability to specify equations.

Inside this issue:

(Continued from page 1)

erty called “qbar-area” that is the product of qbar and reference area, since it is used extensively. Also seen within the product element is a property called, “aero/function/ground-effect-factor-lift”. You can see this referred to in the "aero/coefficient/CLwbh" example farther back. This capability allows for “scratch” functions to be defined and used later one.

```
axis name="LIFT">

  <function name="aero/coefficient/CLwbh">
    <description>Lift due to alpha</description>
    <product>
      <property>aero/function/ground-effect-factor-lift</property>
      <property>aero/qbar-area</property>
      <table>
        <independentVar lookup="row">aero/alpha-rad</<independentVar>
        <independentVar lookup="column">aero/stall-hyst-norm</independentVar>
        <tableData>
                 0.0   1.0
          -0.09 -0.22 -0.22
           0.0   0.25  0.25
           …
           0.32  1.38  0.99
           0.34  1.3   1.05
           0.36  1.15  1.15
        </tableData>
      </table>
    </product>
  </function>
```

Figure 1. Example of how a lift axis coefficient is now specified using a “function” in the JSBSim config file v2.0 format.

Prior to config file format v2.0, JSBSim used “factor groups” to modify coefficients, allowing more complex features to be modeled. For example, when flying close to the ground, coefficients contributing to total lift and drag should be modified by a “ground effect” factor. For v2.0, such a multiplier is defined once in the aerodynamics section of the configuration file – outside of any axis definition:

```
<aerodynamics>

  <function name="aero/function/ground-effect-factor-lift">
    <description>Change in lift due to ground effect factor.</description>
    <table>
      <independentVar>aero/h_b-mac-ft</independentVar> <!-- row lookup -->
      <tableData>
        0.0  1.203
        0.1  1.127
        0.15 1.090
        …
        0.9  1.003
        1.0  1.002
        1.1  1.0
      </tableData>
    </table>
  </function>

  <axis name="LIFT">
  …
```

Figure 2. Specification of a free function (a function not belonging to any axis in particular) in the JSBSim config file v2.0 format.

The property name that will be assigned to the free function above is, “aero/function/ground-effect-factor-lift”. You

The product element (as you might guess) directs JSBSim to take all the parameters specified within it and multiply them together. There are other op-

(Continued on page 3)

News Items

As pointed out in the main article, the new JSBSim config file format is in final preparations for release into the main CVS development branch on the JSBSim web site.

▲

The month of December was a big one for the project, with the highest traffic visiting the web site in the four years that statistics have been kept at SourceForge. Almost 3,500 page views were recorded.

▲

New aircraft models in-work:

Airbus A-380

(Continued from page 2)

erations that can also be specified in the config file:

product	difference
sum	quotient
pow	abs
sin	cos
tan	asin
acos	atan

The operations can be nested, effectively being analogous to the use of parentheses. Operands can be actual values, properties, or the value returned by a 1, 2, or 3 dimensional table:

```
<function>
  <description> example: area of a circle </description>
  <product>
    <pow>
      <sum>
        <property> inner_radius </property>
        <property> outer_radius </property>
      </sum>
      <value> 2.0 </value>
    </pow>
    <value> 3.1416 </value>
  </product>
</function>
```

The syntax of the construct above is very similar to (and was inspired by) MathML, but the format settled on for JSBSim is a smaller subset.

Functions are used in several places (and may show up in the flight control system portion in the near future) and offer a new and powerful capability to JSBSim in modeling aircraft flight dynamics.

New Feature: Specification of Units

Units can now be specified for most items in the configuration file, and new conversions can be added in FGXMLElement.cpp when needed. The basic syntax is this:

```
<parameter unit="UNIT"> </parameter>
```

for example:

```
<wingarea unit="FT2"> 174 </wingarea>
```

In this case, the example shows that the wing area is 174 square feet. The fol-

(Continued on page 4)

The syntax of the [equation] construct … is very similar to (and was inspired by) MathML, but the format settled on for JSBSim is a smaller subset.

(Continued from page 3)

lowing specification would result in the same results (giving the wing area in square meters):

```
<wingarea unit="M"> 16.16 </wingarea>
```

If a given conversion is not handled, an error message will be displayed and the program will exit.

autopilot altitude hold function state, along with support code such as property binding functions for the parameter. Now, specific hardcoded parameters have been removed in many cases. What is available now is the ability to specify a property to be created that is read/write. For instance, in the experimental autopilot definition for the C172x, properties can be caused to be created at runtime as shown in Figure 3.

The new configuration file format and associated files/functions may make their way into the main JSBSim branch in CVS in the next few weeks.

```
<autopilot name="C-172X Autopilot">

<!-- INTERFACE PROPERTIES -->

  <property>ap/attitude_hold</property>
  <property>ap/altitude_hold</property>
  <property>ap/heading_hold</property>
  <property>ap/altitude_setpoint</property>
  <property>ap/heading_setpoint</property>
  <property>ap/aileron_cmd</property>
  <property>ap/elevator_cmd</property>

  …

  <component name="Roll AP Autoswitch" type="SWITCH">
    <default value="0.0"/>
    <test logic="AND" value="fcs/roll-ap-error-summer">
      ap/attitude_hold == 1
    </test>
  </component>
```

Figure 3. Interface properties. The attitude hold property is created as an interface to the "outside", and is used on the "inside" by the component shown.

New Feature: Read/Write Interface Property Creation

In the flight control system model, oftentimes a switch will need to be thrown or a flag set in order to activate a particular control law path. For example, turning on an altitude hold autopilot function required the JSBSim flight control class (FGFCS) to have a hardcoded integer parameter that represented the

In this example, the ap/attitude_hold property can be set from either FlightGear or a JSBSim script, and subsequently affect control law execution by referencing this property.

The new configuration file format and associated files/functions may make their way into the main JSBSim branch in CVS in the next few weeks.

In Depth: Using JSBSim with Matlab / Simulink

By D. Wysong, Aerocross Systems

So, you have a Matlab/Simulink model that you want to integrate with JSBSim? Your model can represent any number of things: a high-fidelity actuator or engine model, a navigation system, or a flight control system. Regardless, you would like to use JSBSim to try your model out in the nonlinear, 6DOF world (taking advantage of a free and open source application). Well, I have some good news and some bad news to share with you.

The good news is that you're crazy – just like me! Whether your desire to integrate these two seemingly different applications is fueled by a genuine need or by a compulsion that doctors can't seem to properly medicate, you're in luck. I've done this on a few occasions and am about to let you know how.

(Continued on page 5)

(Continued from page 4)

Ready for the bad news? You'll need your checkbook…

I'll assume that you already have a Matlab/Simulink license. For those who don't have the resources for a license, be advised that there are open-source alternatives available (Scilab/Scicos/ TrueTime).

The easiest, most pain-free way to integrate Matlab/Simulink with JSBSim (or any other external process) is with a Simulink add-on called Real-Time Workshop (RTW). It isn't the only way to do it, but it certainly is the quickest and cleanest. I will defer to the MathWorks website for complete details about RTW (www.mathworks.com), but the many automotive, UAV, commercial/military aircraft, and spacecraft programs that utilize the functionality offered by Matlab/Simulink/RTW are evidence of the power that these software packages offer. If you buy it you won't be disappointed.

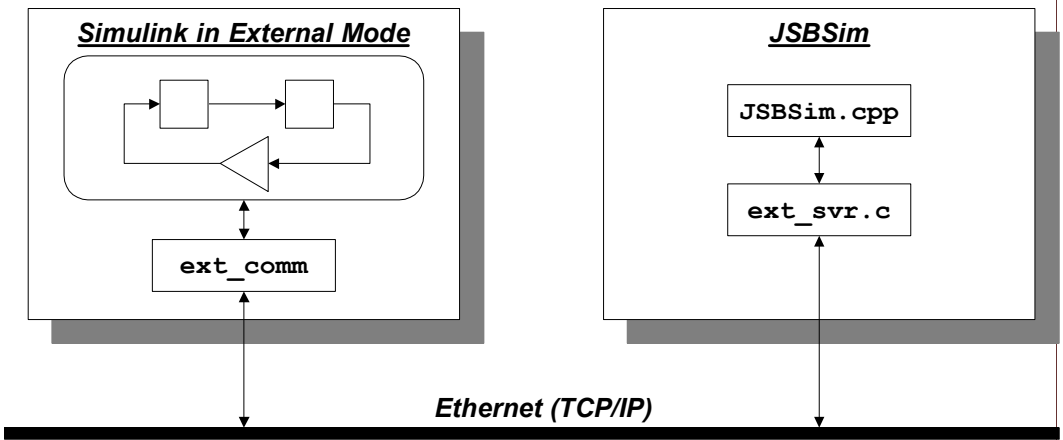


Figure A. Matlab/Simulink integration with JSBSim.

Ok, enough with the sales pitch. What we're going to construct is a simple network between your Simulink model (the client) and your JSBSim (the server). Your Simulink model will communicate with JSBSim via an Ethernet socket, and the new simulation loop will be closed when both sides of the Ethernet link are operating in data-driven, lock-step mode with each other. See Fig. A. for a diagram of what the new Simulink/JSBSim closed-loop 6DOF platform will look like.

Your first task will be to configure your Simulink model for networked operation. This is all made possible through the magic of RTW with something called "External Mode". Behind that fancy name lies the "hooks" your Simulink model needs so that it can act as a network client. RTW also provides the logic that sits behind those ext_* blocks. Rather than add to the confusion, I'm going to step back and raise the old "this is an exercise left to the reader" flag and send you immediately to the MathWorks website and your stack of RTW documentation to learn all there is to know about "External Mode" operation. Yes, that was a dirty trick, but you'll learn more from a half-hearted web surfing attempt than you would from reading chapters of my senseless babble. Trust me.

Under normal circumstances, the next thing you would have to tackle is figuring out how you are going to connect to and manage the network transport layer between your Simulink model and JSBSim. Well, lucky for you these are not normal circumstances! RTW is going to handle all of the hassles commonly associated with those pesky network interfaces including socket initialization, connection management, and socket data handling. The functions and libraries are all provided and it makes the whole ordeal relatively painless. Your only network-related responsibility is to make sure that your data structures – the struct's or union's that you create to hold your input and output data – are consistent on both ends of

(Continued on page 6)

(Continued from page 5)

the network interface. Pass your output data pointer to the relevant RTW "set" function and off it goes. Pass your input data pointer to the relevant RTW "get" function and data appears. See – who said you couldn't get your money's worth from a commercial software package?

Now, what are you going to do about JSBSim? You have a huge pile of documentation that'll help you figure out Matlab/Simulink/RTW, but you're absolutely empty handed when it comes to converting JSBSim into the server for your new simulation network. When stuck in a situation like this there's only one thing you can do – break something!

It isn't quite as fun as it sounds because the ritual violence is limited to one snippet of code. Peer inside the stand-alone JSBSim main program (JSBSim.cpp) and locate the magical while() loop. See that call to FDMExec->Run()? The area above that is going to be the new home for all of your network input/output and data-driven simulation logic.

Here is the pseudo-code for your new, JSBSim server (JSBServer.cpp?):

```
main()
{
  // initialize Ethernet interface
  // using RTW

  result = FDMExec->Run();

  while (result) {

    // LOAD JSBSim output data into
    // Simulink input data
    // structure here.

    // PASS data across socket using
    // RTW here.

    // AWAIT receipt of data on
    // socket using RTW here.

    // PARSE Simulink output data
    // into JSBSim input variables
    // here.

    result = FDMExec->Run();

  }
}
```

After initialization, the `while()` loop is entered and the Simulink input data structures are filled with JSBSim state data. [You need only provide code that implements low-level communications. You need not be concerned with issues such as data conversions between host and target, or with the formatting of messages. Code provided by Real-Time Workshop handles these functions. **http://www.mathworks.com/access/helpdesk/help/toolbox/rtw/rtw_ug/data_ex9.html]**

Next, the input data structures are sent across the network to the Simulink model using RTW functions. JSBSim execution should then hang on a socket read while the Simulink model executes and loads data into JSBSim input data structures. When these data packets arrive on the socket, JSBSim execution continues. The input data packets are parsed and the data are loaded into the appropriate JSBSim variables. Finally, after the logic that I omitted for clarity is executed, FDMExec->Run() is called and the fun starts all over again. Pretty simple, huh?

Notice that RTW is there again to rescue you with source code for initializing the Ethernet sockets, initiating communication with the Simulink client, and sending/receiving data across the network (ext_svr.c, ext_svr_transport.c). As long as you manage the structures for your input/output data and pay attention to the logic sequence, RTW will handle the rest of the networking nastiness.

So, there you have it – how to integrate your Matlab/Simulink models with JSBSim. Keep in mind that I have only exposed you to one of the many options that RTW makes available. You could also use RTW to generate C source code and directly integrate that with JSBSim. The pseudo-code remains the same, you just get to ignore all of the networking nonsense and make C function calls instead. Have fun!

Integrating JSBSim with Your Application

By Jon S. Berndt

JSBSim is meant to be a self-contained, platform-independent, easy-to-incorporate flight dynamics model (FDM) that can be leveraged by developers in crafting their own simulation applications. Unfortunately (as often happens in open source projects) documentation is one of the last things to be developed, and so new users can be overwhelmed. This article will (hopefully) help new developers get started by pointing out the documentation that *is* currently available, and by instructing the reader how JSBSim can be incorporated into an application.

For an overview on JSBSim that contains some insight into the architecture and how an external application might use JSBSim, see the JSBSim web site, www.jsbsim.org. Select the Documentation link. Some of the articles are outdated – and more are destined to be outdated soon as the config file format is changing – but these articles should be most helpful:

1. AIAA-2004-4923 "JSBSim: An Open Source Flight Dynamics Model in C++"
2. Draft JSBSim API *Documentation* (see the documentation page at www.jsbsim.org, as well as the documentation for FGFDMExec)

The next place to look closely would be the interface code used to splice JSBSim with FlightGear, JSBSim.cxx. This code is managed in CVS (the source code repository) at www.jsbsim.org. The code in JSBSim.cpp – the *stub* code used to instantiate JSBSim in a standalone capability – is also instructive.

Using JSBSim can be summarized in the order of execution:

1. Instantiate FGFDMExec
2. Load the aircraft model
3. Load the initial conditions
4. Trim at the initial conditions (not absolutely required, but recommended)
5. Begin cyclic execution (call the Run method of the FGFDMExec class)

One can also run JSBSim using a script, which takes care of loading the aircraft and initial conditions that are specified in the script file. The operation using scripts is almost identical:

1. Instantiate FGFDMExec
2. Instantiate an FGScript object
3. Load the script (the script will specify and load the aircraft and initial conditions)
4. Cyclically call the script object's RunScript method and the Run method of the FGFDMExec class)

This part of the process is all fairly straightforward. In the case of scripted operation, pilot commands or inputs to the control system or autopilot are controlled by the script itself. Outputs can also be sent over a socket connection by proper specification in the configuration file. But, what about the case where JSBSim is used in an application and scripting is not utilized? How does one input commands and receive state information as output (i.e. velocity, position, attitude, etc.)?

There are two mechanisms to do this. One is by using class methods themselves to directly set or get the relevant parameters. The other is to use the property system to do so. Looking at the FlightGear interface to JSBSim (FGJSBSim in JSBSim.cxx) one can see that the approach used there is the former. An excerpt from the class is presented here (heavily edited to show only the needed functionality):

```
void FGJSBsim::update(  )
{
  copy_to_JSBsim();
  fdmex->Run();
  copy_from_JSBsim();
}
```

The copy_to_JSBSim() function does the following (again, heavily edited):

```
bool FGJSBsim::copy_to_JSBsim( )
{
  // copy control positions
  // into the JSBsim structure

  FGFCS* FCS=FDMExec->GetFCS();
```

(Continued on page 8)

Highlighted References

Online:

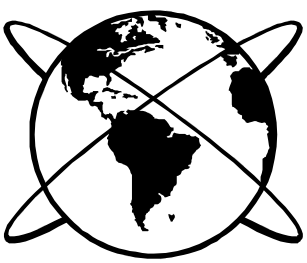
Custer Channel Wing
www.custerchannelwing.com

http://techreports.larc.nasa.gov/ltrs/PDF/2002/aiaa/NASA-aiaa-2002-3275.pdf

Helicopter Modeling
www.robertheffley.com/docs/Manudyne%2083-2-3.zip

www.simlabs.arc.nasa.gov/library_docs/rt_sim_docs/Toms.pdf

Visit us on the web at: www.jsbsim.org



(Continued from page 7)

```
  FCS->SetDaCmd(local_ail_cmd);
  FCS->SetRollTrimCmd
                (local_ail_trim_cmd);
  FCS->SetDeCmd(local_elv_cmd);
  FCS->SetPitchTrimCmd
               (local_elev_trim_cmd);
  FCS->SetDrCmd(local_rud_cmd);
  FCS->SetYawTrimCmd
                (local_yaw_trim_cmd);
…
```

The flight control system class FGFCS is the interface to pilot commands, and methods exist for setting the control commands in that class. Since the interface class knows the executive class instance (FGFDMExec is instantiated from within the interface class FGJSBSim) and since there are methods in the executive class that get pointers to the instances of the model classes (such as FGFCS) one can gain access to the specific control setting functions.

Getting state information from JSBSim is pretty much just the reverse process:

```
bool FGJSBsim::copy_from_JSBsim()
{
  set_local_Accels_Body(
    Aircraft->GetBodyAccel(eX),
    Aircraft->GetBodyAccel(eY),
    Aircraft->GetBodyAccel(eZ)
  );

  set_local_Velocities_Local(
    Propagate->GetVel(eNorth),
    Propagate->GetVel(eEast),
    Propagate->GetVel(eDown)
  );
  …
```

We'll leave the other access method (the use of properties) for a later issue.

Simulate This! The Custer Channel Wing

Exercise: Design a simple (without even flaps), 450 HP, five passenger plane, capable of slow flying at 20 mph, 160mph cruise, 200 foot takeoff and landing run, with extreme load carrying ability.

Sometime in the 1920s, Willard Custer took shelter in a barn during a hurricane. Much to his surprise and fascination, the roof of the barn suddenly lifted off, and soared through the air. He wondered why an airplane had to gather speed on a runway, while a barn roof, a poor airfoil by any reckoning, could fly from a standing start. He soon came to the realization that it was the speed of the air over the surface, not the speed of the surface through the air that created lift. Bernoulli's principle applied in both cases. He settled on the idea of pulling the air through channels that were, in fact, the lower half of a venturi. He was reversing the normal method of powered flight. Instead of using the engines to move the airfoil through the air, he used the engine to move the air through the airfoil. His channel had the effect of going several hundred miles per hour, due to the induced air flow, while standing still. The airflow over the surface of the channel created conventional lift, and a lot of it. It was at this point that Custer settled on," It's the speed of the air, not the airspeed", which became his mantra of "aerophysics".

About this newsletter …

"Back of the Envelope" is a new communication tool written for a wider audience than core JSBSim developers, including instructors, students, and other users. The articles featured will likely tend to address questions and comments raised in the mailing lists and via email. If you would like to suggest (or even author) an article for a future issue, please email the editor at: jsb@hal-pc.org.